PRV

PATENT- OCH REGISTRERINGSVERKET
Patentavdelningen

Intyg
Certificate

Härmed intygas att bifogade kopior överensstämmer med de handlingar som ursprungligen ingivits till Patent- och registreringsverket i nedannämnda ansökan.

This is to certify that the annexed is a true copy of the documents as originally filed with the Patent- and Registration Office in connection with the following patent application.

(71) Sökande   Telefonaktiebolaget L M Ericsson (publ), Stockholm
Applicant (s)   SE

(21) Patentansökningsnummer   0401850-3
Patent application number

(86) Ingivningsdatum   2004-07-08
Date of filing

(30) Prioritet begärd från   2003-12-19 SE 0303497-2

Stockholm, 2004-12-28

För Patent- och registreringsverket
For the Patent- and Registration Office

Gunilla Larsson

Avgift
Fee

PRIORITY DOCUMENT
SUBMITTED OR TRANSMITTED IN
COMPLIANCE WITH
RULE 17.1(a) OR (b)

1

IMAGE PROCESSING

TECHNICAL FIELD

The present invention generally refers to image processing, and in particular
to methods and systems for encoding and decoding images.

BACKGROUND

Presentation and rendering of images and graphics on data processing
systems and user terminals, such as computers, and in particular on mobile
terminals have increased tremendously the last years. For example, three-
dimensional (3D) graphics and images have a number of appealing
applications on such terminals, including games, 3D maps and messaging,
screen savers and man-machine interfaces.

A 3D graphics rendering process typically comprises three sub-stages.
Briefly, a first stage, the application stage, creates several triangles. The
corners of these triangles are transformed, projected and lit in a second
stage, the geometry stage. In a third stage, the rasterization stage, images,
often denoted textures, can be "glued" onto the triangles, increasing the
realism of the rendered image. The third stage also performs sorting using a
z-buffer.

However, rendering of images and textures, and in particular 3D images and
graphics, is a computationally expensive task in terms of memory bandwidth
and processing power required for the graphic systems. For example,
textures are costly both in terms of memory, the textures must be placed on
fast on-chip memory, and in terms of memory bandwidth, a texture can be
accessed several times to draw a single pixel.

In order to reduce the bandwidth and processing power requirements, an
image (texture) encoding method or system is typically employed. Such an
encoding system should result in more efficient usage of expensive on-chip
memory and lower memory bandwidth during rendering and, thus, in lower

power consumption and/or faster rendering. This reduction in bandwidth and processing power requirements is particularly important for thin clients, such as mobile units and telephones, with a small amount of memory, little memory bandwidth and limited power (powered by batteries).

Textures with an alpha component enable semi-transparent or punch-through textures, which are used in a wide range of applications. A texture compression system that allows for alpha textures is thus of great importance.

Delp and Mitchell [1] describe a simple scheme, called block truncation coding (BTC), for image encoding. Their scheme encodes gray scale images by considering a block of 4 pixels × 4 pixels at a time. For each such block, two 8-bit gray scale values are generated and each pixel in the block then uses a single bit to index one of these gray scales. This results in a compression rate of 2 bits per pixel (bpp). However, BTC suffers from production of artifacts, above all in regions around edges and in low contrast areas containing a sloping gray level. Furthermore, edges in a gray scale image processed by BTC have a tendency to be ragged.

A simple extension of BTC, called color cell compression (CCC), was presented by Campbell et al. [2]. Instead of using two 8-bit gray scale values for each image block, two 8-bit values are employed as indices into a color palette. This palette comprises 256 colors represented by 8 bits for each of the R, G and B component. An 8-bit index then points at a (24-bit) color in the palette. This allows for compression of images at 2 bpp. However, this does require a memory lookup in the palette. In addition, the palette is restricted in size. The CCC scheme also introduces large color "jaggies" and poorly encodes the case where more than two colors exist in an image block.

In a patent description [3], Iourcha et al. disclose a texture compression scheme called S3TC (S3 Texture Compression) or DXTC (DirectX Texture Compression), that can be seen as an extension of CCC. An image is

decomposed into a number of image blocks of 4 pixels × 4 pixels. Each such image block is encoded into a bit sequence of 64 bits, thus resulting in a compression rate of 4 bpp. The 64-bit sequence comprises two basic colors or color codewords (16 bits each) and a 32-bit sequence of 2-bit indices, one index for each pixel in the image block. During decoding, a color palette of four colors is generated. The first two RGB (red, green and blue) colors of the palette correspond to the two basic colors (codewords). The two additional colors, situated between the basic colors in the RGB space, are then interpolated therefrom. Each 2-bit index then identifies, for each pixel, one of the four colors of the palette to use for that pixel.

Although, the S3TC scheme works fairly well for computer terminals, it is not well adapted for mobile units and other thin clients. Such mobile units typically only have memory busses of 16 or 32 bits at best. Thus, at least two, and possibly up to four, memory accesses are required to read out the 64-bit compressed version of an image block, if S3TC is implemented in a mobile unit. In addition, during the interpolation of the two additional colors of the color palette, multiplication by 1/3 and 2/3 is performed, which is not ideal in hardware. The compression using S3TC is also relatively time consuming, at least on a mobile terminal.

The S3TC scheme handles alpha textures as a special case. The bits are then interpreted differently in order to allow for punch-through alpha (each pixel being either completely opaque or transparent). The fact that S3TC cannot handle more accurate alpha is one of the major drawbacks of the system.

Akenine-Möller and Ström [4] have developed a variant of S3TC, called POOMA, which is specifically targeted for mobile phones. In POOMA, an image block of 3 pixels × 2 pixels is encoded into 32 bits, giving 5.33 bpp. The encoded 32-bit representation of an image block is adapted for the memory busses of mobile phones, which typically are 32 bits at best. Thus, a pixel can be rendered using only one memory access compared to two accesses for S3TC. POOMA uses two base colors but only one additional

color interpolated between the base colors, resulting in a color palette of three colors.

A major disadvantage of POOMA is the block size of 3 × 2 pixels. As a result, calculation of the block address for a particular pixel or texel (texture element) requires division by 3, which is not ideal for hardware design. Furthermore, the widths and heights of textures (images) in graphics are typically always a power of two, which means that a block width of 3 is inconvenient. As for S3TC, the encoding using POOMA is relatively time consuming, in particular when implemented on a mobile terminal.

Fenny [5] discloses an image-encoding scheme used in the MBX graphics hardware platform for mobile phones. This scheme uses two low-resolution images, where one image is usually a low-pass filtered version of the original image. During decoding, bilinear magnification (upscaling) of these two images is performed. Each pixel also stores a blend factor between these two upscaled images. 64 bits are used for encoding each image block and compression rates of 2 bpp and 4 bpp are described. Information from neighboring image blocks is needed, which complicates decoding.

DETAILED DESCRIPTION

The present invention relates to image and graphic processing, and in particular to encoding or compressing images and decoding or decompressing encoded (compressed) images.

Generally, according to the invention, during image encoding, an image is decomposed or divided into a number of image blocks. Each such image block then comprises multiple image elements having, among others, a certain color. The image blocks are then encoded to generate an encoded representation of the image.

When an encoded image or graphic primitive subsequently is to be rendered, e.g. displayed on a screen, the relevant image elements of the encoded image

blocks are identified and decoded. These decoded image elements are then used to generate a decoded representation of the original image or graphics primitive.

5   The present invention is well adapted for usage with three-dimensional (3D) graphics, such as games, 3D maps and scenes, 3D messages, e.g. animated messages, screen savers, man-machine interfaces (MMIs), etc., but is not limited thereto. Thus, the invention could also be employed for encoding other types of images or graphics, e.g. one-dimensional (1D) or two-
10   dimensional (2D) images.

       In 3D graphics processing, typically several triangles are created and the corresponding screen coordinates of the corners of these triangles are determined. Onto each triangle, an image (or portion of an image), or a so-
15   called texture, is mapped ("glued"). The management of textures is, though, costly for a graphic system, both in terms of utilized memory for storage of textures and in terms of memory bandwidth during memory accesses, when textures are fetched from the memory. This is a problem particularly for thin clients, such as mobile units and telephones, with limited memory capacity
20   and bandwidth. As a consequence, a texture or image encoding scheme is often employed. In such a scheme, a texture is typically decomposed or divided into a number of image blocks comprising multiple texels. The image blocks are then encoded and stored in a memory. Note that the size of an encoded (version of an) image block is smaller than the corresponding size of
25   the uncoded version of the image block.

       In the present invention the expression "image element" refers to an element in an image block or encoded representation of an image block. This image block, in turn, corresponds to a portion of an image or texture. Thus, according to the invention, an image element could be a texel (texture
30   element) of a (1D, 2D or 3D) texture or a pixel of a (1D, 2D or 3D) image. Generally, an image element is characterized by certain image-element properties, such as a color value. Furthermore, in the following, the term

6

"image" is used to denote any 1D, 2D or 3D image or texture that can be encoded and decoded by means of the present invention, including but not limited to photos, game type textures, text, drawings, etc.

5   In the present invention, the PACKMAN image compression [6] developed by the same inventors as for the present invention is extended in order to enable alpha images and textures.

In order to simplify understanding of the present invention, a brief summary
10   of the PACKMAN scheme follows below.

In PACKMAN [6], an image to be encoded is decomposed into a number of image blocks comprising multiple image elements (pixels or texture elements, texels). An image block preferably comprises eight image elements
15   and has a size of $2^m \times 2^n$ image elements, where $m=3-n$ and $n=0, 1, 2, 3$. Each image element in a block is characterized by a color, e.g. a 12-bit RGB (red, green, blue) color. The image blocks are then encoded.

In this lossy block encoding, a color codeword is determined for the image
20   block. The color codeword is a representation of the colors of the image elements of the image block. A preferred representation is an average value of the colors of the image elements in block, quantized into 12 bits (4 bits for each of the three color components for a RGB color). Thus, one and the same color codeword (i.e. color representation) is generated for an image block, i.e.
25   for all the image elements of the image block. Thereafter, an intensity codeword is provided for the image block. This intensity codeword is a representation of a set of multiple intensity modifiers that are used (during decoding) for modifying or modulating the intensity of the image elements in the image block. Once the intensity codeword is provided, intensity representations for the image elements in the image block are selected. Each such intensity representation is associated with an intensity modifier from the intensity modifier set. In other words, the intensity representation allows

identification of which intensity modifier from the set to use for a specific image element of the block.

The resulting encoded image block then comprises the color codeword, preferably 12 bits, the intensity codeword, preferably 4 bits, and a sequence of the intensity representations, preferably 8×2=16 bits. The resulting size of an encoded image block is, thus, only 32 bits and a compression rate of 4 bits per pixel (image element) is obtained. This small 32-bit size is well adapted for thin clients, such as mobile units and telephones, which typically have memory busses of 16 or 32 bits. As a result, only one or at worst two memory accesses are then needed to read out the encoded image block from a memory location. A possible bit layout of the 32 bits is shown in Fig 9.

In a preferred embodiment of PACKMAN, the intensity codeword is an intensity index allowing identification of an intensity modifier set. This index could then identify or point to the set in a table or codebook comprising several different intensity modifier sets. Each set preferably comprises four (mathematically) complementary modifier values. In such a case, the modifier sets stored in the table only have to comprise two different intensity modifier values each and then the other two (complementary) values of the set could be calculated therefrom. In addition, the intensity table preferably comprises both sets that include small intensity modifier values, which are adapted for allowing representation of smoothly changing surfaces, and sets that include large intensity modifier values, which are adapted for allowing representation of sharp edges.

During decoding, the encoded image block(s) that should be decoded is identified and fetched from e.g. a memory location. Once the correct encoded image block is identified an intensity modifier set is provided. This modifier set is provided based on the intensity codeword in the encoded image block. This set provision is preferably performed by identifying, by means of the intensity codeword, an intensity modifier set from an intensity table comprising multiple modifier sets.

Thereafter, a color representation is generated for at least one of the image elements of the image block. This color generation is performed based on the color codeword in the encoded block representation. The intensity modifier to use for the image element that should be decoded is then selected. The

5   modifier value is selected from the provided modifier set based on the intensity representation associated with the image element and found in the representation sequence of the encoded image block. Once the correct intensity modifier value is selected, the intensity of the image element is modified with this value.

10

The selection of intensity modifier and modification of the intensity are preferably performed for all image elements that should be decoded in the current encoded image block. The block decoding is then preferably repeated for all image blocks that comprises image elements that should be decoded.

15   Thereafter, a decoded representation of an original image, or a portion thereof, can be generated based on the decoded image elements and blocks.

The color representation is preferably generated by expanding the three 4-bit color components of the color codeword into three 8-bit components. The

20   resulting 24-bit color is then assigned to the image element(s) of the image block that are to be decoded. The intensity of the image elements is preferably modified by adding or multiplying the intensity modifier to each color component, or each color component is otherwise modulated with the intensity modifier. Thereafter, the resulting intensity modified color

25   components are clamped between a minimum and maximum threshold value.

Fig. 1 illustrates a hardware implementation of a block decoder or decompressor according to PACKMAN.

In the present invention, different operation modes are used by the compression scheme depending on whether the image is alpha free or it is an

alpha image. Two different embodiments of the invention will now be described.

In the first embodiment, three new operational modes are defined, in order to allow for alpha textures and images. In the following, these modes are defined as PACKMANalpha-PACKMAN (or PAP) mode, PACKMANalpha-ALPHA (or PAA), and the PACKMANalpha-TWOTIMER (or PAT) mode.

When compressing an image, it is first investigated whether there is any alpha information at all in the entire texture or image. If the image is an RGB image (where alpha is implicitly 255 everywhere), or if it is an RGBA image where the alpha component is set to 255 in every image element of the image, we say that the image is alpha-free, or that it is a PACCCKMAN image and we encode the image using the traditional PACKMAN scheme [6] or an enhanced PACKMAN scheme denoted PACCCKMAN, described briefly herebelow.

The PACCCKMAN scheme results in an increased image quality, while a relatively low hardware complexity is kept. The basic idea is to combine a modified form of PACKMAN with modified form of CCC [2]. The new scheme contains two different operation modes, PACCCKMAN-PACKMAN (PP for short, based on PACKMAN), and PACCCKMAN-CCC (PC for short, based on CCC).

Thus, in PACCCKMAN each 2x4 texel (image) block is still compressed separately into 32 bits. However, depending on the contents of the block, we may choose to either use PC or a PP. One bit, e.g. the first bit of the 32 bits, is therefore reserved for indicating which of PP and PC is used to compress that block. For best results, one should obviously choose the technique that gives least difference when compared to the original image. We call this bit, the *modebit*, and if modebit=1, then PC is used, and otherwise PP is used. It is anticipated by the invention, that a modebit=0 can alternatively be used to represent PC. In such a case, PP is associated with the modebit=1.

Herebelow, the PACCCKMAN-PACKMAN (PP) scheme and the PACCCKMAN-CCC (PC) scheme are described.

PACCCKMAN-PACKMAN (PP) mode

Since the original PACKMAN scheme used all 32 bits, we need to simplify the scheme so that only 31 bits are used – the remaining bit is the modebit. This can be done by using only 3 bits for the intensity codeword. Thus, the table or intensity codebook will only contain 32 values instead of 64 values, and this modification therefore makes the hardware decompressor simpler. The actual intensity modifier values in the codebook may need to be recomputed and optimized compared to PACKMAN [6].

PACCCKMAN-CCC (PC) mode

CCC is modified for our purposes as follows. Two base colors or color codewords are stored, and each base color could be stored using e.g. 4+4+4=12 bits. Each of the 2x4 image elements are associated with one bit, as an index into one of the two base colors. The total sum will then be 12+12+8=32 bits. However, this is one bit too many, since we also need to store the modebit. This problem is solved by assuming that the one of the image elements of the block, preferably the first image element, always contains the first (or the second) base color (if that is not the case, then we can swap base color 0 and base color 1). In this way, we do not need an index bit for the first image element, and all fits in 32 bits again.

The advantage is that the scheme is still extremely simple, and it can give an additional boost in image quality of 1 dB, which is very significant. There are two main novelties of PACCCKMAN. The first novelty is to combine two different image compressing schemes, in this case PACKMAN compression with CCC. This has not been done before. The second novelty is the way we encode pixel color, using only seven bits instead of eight. Furthermore, PACCCCKMAN compensates for the relative inability of regular PACKMAN to handle blocks with two distinct colors of similar luminance. Even though

PACCCKMAN is slightly more complex than PACKMAN, the extra complexity is extremely small, which means that PACCCKMAN is easily implemented in both software and hardware.

However, if, during the initial image investigation, it is determined that there exists one image element or more in the image that does not have an alpha of 255 (or some other predefined, preferably fixed, value), we say that the image is an alpha-image. In this case, one of the three new modes PAP, PAA or PAT is used when compressing the image. Note that even though several of the blocks in the image might not contain alpha, the image is still, in this embodiment, an alpha image if there exists at least one image element that has an alpha other than 255.

Note that we use (store) one bit for the entire image for discriminating whether the image is alpha free (a PACKMAN or PACCCKMAN image), or whether it is an alpha image (a PACKMANalpha image).

PACCCKMAN image (PP and PC)
If the image is a PACCCKMAN image (alpha free), all blocks are compressed according to the PACCCKMAN scheme described above. Fig. 2 illustrates the bit layout for PP and PC, respectively.

PACKMANalpha image (PAP PAA or PAT)
If the image is an alpha image, the block is of one of the new modes PAP, PAA or PAT. Their bit layouts are in Fig. 3.

PACKMANalpha-PACKMAN block (PAP)
In all three modes, the first bit is a mode bit. If this bit is zero, the block will be a PAP block, and will not contain alpha information. The PAP block is treated just as a PP block.

If the mode bit is one, the block will contain transparency, and will either be of the PAA type or the PAT type. If bit4 = bit12 AND bit8=bit16 (see Fig. 3),

then the block is of PAT type, else it is of PAA type. We will first describe how to decompress a PAA block.

PACKMANalpha-ALPHA block (PAA)

If the block is of PAA type, we cannot afford to spend twelve bits on the general color of the block. Instead we only use 3 bits per R, G and B encoded color component. In order to simplify hardware, we put the three bits of R in the same place as the three most significant bits in the PP format. The same goes for G and B. Hence we get a few leftover bits: bit4 ($\alpha A0$ in Fig. 3), bit8 ($\alpha A1$), and bit 12 ($\alpha B0$). Together with bit16 ($\alpha B1$), these bits make up two 2-bit alpha values: αA ($\alpha A0$, $\alpha A1$) and αB ($\alpha B0$, $\alpha B1$). Since αA is two bits wide, it can assume the transparency values 0.0, 0.3333, 0.6667 and 1.0, (0, 84, 170 and 255 in 8-bit notation) and the same goes for αB. A person skilled in the art will understand that it is possible to let these bits represent other alpha values, for instance, 0.0, 0.5, 0.75 and 1.0.

Each image element in the block can assume either αA or αB, and the bits 17 through 23 are used to select which value should be assigned. The exception is the first pixel, which is in this embodiment always assigned αA. Other solutions could alternatively be employed. In this way, only seven bits are needed for the bit mask. Note that the alpha information is encoded in a CCC manner.

The color information of the block is decoded in much the same way as in PACCCKMAN-PACKMAN (PP), except for two differences. The first difference is that the R, G and B values are stored in 3 bits only, which means that they will have to be expanded from three to eight bits before the luminance is additively modified. This is done by just repeating the three bits until the 8-bit value is filled. For instance, the value 001 will become 00100100. The second difference is that there are not enough bits to let the luminance be modified on a per-pixel basis. For, this, 16 bits would be required, but only eight remain. Instead, we share the color information between two image

elements, effectively reducing the resolution for the color information to half in on direction, as shown below:

Normal Block

A	E
B	F
C	G
D	H

Half resolution block

AB	EF
CD	GH

Note though, that he alpha information can change for every image element in the block – it is thus only for the color information that the resolution is halved. Thus, instead of having to store luminance information (intensity representation sequence) for eight image elements (A, B, C, D, E, F, G and H), we only need to store luminance information for four image elements or pixels (pixel AB, pixel CD, pixel EF and pixel GH). Thus only eight bits are needed. Bit24 will be most significant bit (MSB) for pixel AB, bit25 will be least significant bit (LSB) for pixel AB, bit 26 will be MSB for pixel CD, and so forth.

The halving of the resolution will of course have an impact on image quality. However, it is only in the blocks where we have varying alpha information that this happens. Many of the blocks of the image will have a constant alpha of 255, and can be coded using the PAP mode, with high quality. Other blocks will have a constant alpha different from 255, and can be coded using the last mode (PAT) presented below.
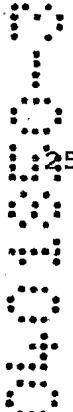
PACKMANalpha-TWOTIMER block (PAT)

Note that in the PAA mode, it does not make sense to have αA and αB assuming the same value, for instance 0.3333. If we would like that all image elements should have the value 0.3333, we could set αA to 0.3333 and then set the alpha index bits 17 through 23 to zero. Thus we can use αA = αB to signal a special mode, which we call PACKMANalpha-TWOTIMER (PAT for short). Thus, if bit4 = bit12 AND bit8=bit16, then the block is of PAT

14

type and the decompression should be done according to the description below.

The general color (color codeword) is stored in the same way as for a PAA
5   block, i.e., 3 bits for each R, G and B component in bits 1-3, 5-7 and 9-11.
Bit4 always equals bit12 and therefore does not carry any information. The
same is true for bit8, it mirrors bit16. Bit12 carries information about the
alpha value for all the image elements in the block. Thus, all image elements
of a PAT block has the same alpha. If bit12 = 0, then all image elements
10 could have alpha value 0.3333 (or, for instance, 0.5). If bit12 = 1, then all
image elements could have alpha value 0.6666 (or, for instance, 0.75). (Note
that having all alpha values equalling zero means that the block is
completely transparent, and therefore the PAA mode will manage the block
perfectly. Note also that if all alpha values are 1.0, this means that the PAP
15 mode can be used for the block.)

The luminance information is obtained in the same manner as for a PAP
block. The only difference is that R, G and B are of three bits instead of four
and needs to be expanded to eight in the same way as for the PAA block.
20 Contrary to the PAA block, there is full per-pixel information for the
luminance in bit 16 through 31, i.e. the intensity representation sequence
includes 16 bits.

We will now show how decompression works. Decompression using PP and
25 PC mode was briefly described above and is not further discussed. Since the
PAP mode is equivalent with the PP mode, we will not go through that.

<u>Decompression of a PAA block</u>
In the following example a 32 bit input word of $fa51367a_{hex}$ = 1 111 1 010 0
30 101 0 001 0 0110110 01 11 10 10$_{bin}$ of the form (mode bit)(R 3 bits)(αA0)(G 3
bits)(αA1)(B 3 bits)(αB0)(intensity codeword 3 bits)(αB1)(alpha index 7
bits)(luminance index or intensity representation sequence 2 bits times 4) is

assumed. However, this is merely an example and the invention is not limited thereto.

First, we investigate the modebit, to find out that it equals 1. Thus it is a PAA or PAT block. Next, we see whether bit5=bit12 and bit8=bit16. However, since bit5=1 and bit12=0 this is not the case and we can confirm that the block is of PAA type.

Next, we decode the alpha values αA and αB. $(\alpha A0, \alpha A1)=(1,0)$, and thus $\alpha A=10_{bin}$. By extending this value to eight bits we get $\alpha A = 10101010_{bin} = 170$. Since 255 equals 1.0, αA should be interpreted as $170/255 = 0.6667$. Doing the same for αB yields $00_{bin} \Leftrightarrow 00000000_{bin} = 000$ and $000/255 = 0.0000$. Now we can decode the alpha values for the individual image elements. The first image element or pixel (pixel A below) always gets αA (170) in this embodiment. We then use bits 17 through 23 to assign alpha values for the following pixels. Using the notation in the block below,

A	E
B	F
C	G
D	H

pixel B has index 0 which means αA (170), pixel C has index 1 which means αB (0), pixel D also gets αB (0) and so forth, until we have the following alpha values:

170	170
170	0
0	0
0	170

Next, we decode the luminance information. We start by extracting the general color of the image block based on the color codeword. The three R-bits are 111_{bin}, which give $11111111_{bin} = 255$ after extension to 8 bits. In the same manner, $G = 010_{bin} \Leftrightarrow 01001001_{bin} = 73$, and $B = 101_{bin} \Leftrightarrow 10110110_{bin} = 182$. Thus the general color is (255,73,182).

We continue by making use of the 3-bit intensity codeword. The value is 001_{bin}, and thus we should use this intensity codeword for finding the correct intensity modifier values from a table or codebook with intensity modifier values, e.g. Table 1 below. (Note that the values of the table below are just example values – these values can be optimized/changed to obtain better image quality.)

Table 1

Set index	0	1	2	3	4	5	6	7
	-8	-12	-31	-34	-50	-47	-80	-127
	-2	-4	-6	-12	-8	-19	-28	-42
	2	4	6	12	8	19	28	42
	8	12	31	34	50	47	80	127

Thus, $001_{bin} = 1$ and we should use the modifier set (-12, -4, 4, 12). Finally, we modify the image elements additively using the index values. Since we have reduced resolution, we use the same index for A and B, the same for C and D and so forth. The index 11_{bin} means –12, 10_{bin} means –4, 00_{bin} means 4 and 01_{bin} means 12. Thus, since the index for pixel AB is 01_{bin}, we should add (12, 12, 12) to the color (255, 73, 182), yielding (267, 85, 194). After clamping (between a minimum pixel value, e.g. 0, and a maximum pixel value, e.g. 255), we get (255, 85, 194). Thus the color value of pixel A and pixel B should be (255, 85, 194). Doing the same for the other pixels yields:

AB: (255,85, 194)	EF: (251, 69, 178)
CD: (243, 61, 170)	GH: (251, 69, 178)

Thus, the final RGBA block is:

(255,85, 194,170)	(251, 69, 178,170)
(255,85, 194,170)	(251, 69, 178,0)
(243, 61, 170,0)	(251, 69, 178,0)
(243, 61, 170,0)	(251, 69, 178,170)

5   ## Decompression of a PAT block

In the following example a 32-bit input word of $fa59367a_{hex}$ = 1 111 1 010 0 101 1 001 00 11 01 10 01 11 10 10$_{bin}$ of the form (mode bit)(R 3 bits)(copy of bit12)(G 3 bits)(copy of bit16)(B 3 bits)(α)(intensity codeword 3 bits)(luminance index, intensity representation sequence 2 bits times 8) is

10   assumed.

First, we read the modebit and discovers that it equals 1. Thus it is a PAA or PAT block. Next, we see whether bit5=bit12 and bit8=bit16. Since bit5=bit12=1 and bit8=bit16=0 this is the case and we can confirm that the

15   block is of PAT type.

A PAT block has the same alpha value for every image element, and it is given by bit12 (α). In our case, that bit is 1, which means that the alpha value is 0.6667 or 170/255. Thus, the alpha values for the block are given

20   by:

170	170
170	170
170	170
170	170

The colors are extracted the same way as in the PAA example, thus the general color is again (255,73,182).

The intensity modifier set is also chosen the same way, and 001bin gives once again the modifier set (-12, -4, 4, 12).

In contrast to the PAA example, we now have one luminance or intensity modifier index per image element or pixel. Pixel A gets the modifier index 00bin which gives the modifier value 4 from the set above. Thus the color of pixel A is (255+4, 73+4, 182+4) = (259, 77, 186), which after clamping yields (255, 77, 186). Pixel B gets index 11bin, giving value −12, yielding (255-12, 73-12, 182-12) = (243, 61, 170), and so forth. Combining this with the alpha data gives the following decoded RGBA block:

(255, 77, 186, 170)	(255, 85, 194, 170)
(243, 61, 170, 170)	(243, 61, 170, 170)
(255, 85, 194, 170)	(251, 69, 178, 170)
(251, 69, 178, 170)	(251, 69, 178, 170)

The other embodiment of the present invention to handle alpha is more limited in terms of alpha – it can only handle alpha of 0.00 or 1.00 (0 or 255).

Just as in the PACKMANalpha system, the image can be either alpha free or contain alpha. If it is alpha free, we say that it is a PACCCKMAN image and we handle it using modes PP and PC, in the same way as described above.

However, if it contains alpha, we say that it is a PACKMAN-punch-through image. In that case, we have two new modes: PACKMAN-Punch-PACKMAN (PPP), and PACKMAN-Punch-Through (PPT). They have the bit layout illustrated in Fig. 4.

The first mode, PPP, is a direct copy of the PP or PAP modes above. The PPP mode should be used for blocks that do not contain alpha (but other blocks in the image do). It is decoded in exactly the same way as PP or PAP.

5   The PPT mode is very similar to the PPP mode. The general color is obtained the same way. The major difference comes in the way the table or codebook of intensity modifier sets is constructed. Instead of having the normal table of sets, where each set has four different entries (modifier values), we have a table according to Table 2 below. (Note again that the numeric values in the

10   table below are just example values and can be optimized/changed for better image quality.)

Table 2

Set index	0	1	2	3	4	5	6	7
	-4	-8	-12	-20	-36	-47	-80	-127
	0	0	0	0	0	0	0	0
	0*	0*	0*	0*	0*	0*	0*	0*
	4	8	12	20	36	47	80	127

15   Thus, if set 1 is selected, we get the set (-8, 0, 0*, 8). If the alpha/luminance or intensity index bits indicates 0*, the output color will be the general color as defined by the color codeword, and alpha will be 0. If the index bits indicate −8, 0, or 8, the output color will be the general color modified with

20   the respective intensity modifier, and the alpha will be 255. For instance, if the general color is (255,73,182), and the index bits indicate −8, we will get the RGBA color (247, 65, 174, 255).

This second embodiment gives more bits to the color codeword (4 bits for each RGB component compared to 3 bits per RGB component) and simpler

25   hardware, but gives worse alpha behavior (only punch-through instead of two bit alpha).

Herebelow, we describe a possible hardware layout of the PACCCKMAN scheme, and that of the combined scheme of embodiment 1 (PACKMANalpha) and embodiment 2 (PACKMAN-punch-through). This should merely been seen as illustrative embodiments and the present invention is not limited thereto.

Hardware layout for PACCCKMAN
Fig. 5 illustrates the hardware layout for PACCCKMAN.

Hardware layout for PACKMANalpha
Fig. 6 illustrates a hardware layout for PACCCKMAN combined with PACKMANalpha. Note that much of the logic is used for all five modes, giving a hardware design of low complexity. Of course, all the logic that handles the alpha component (lower right corner) has been added. Furthermore, due to the larger number of modes, we have more control logic. From the "Mode determiner" in the top left corner, signals are propagating throughout the hardware design. Still, the design is very small given that it encompasses and entire image compression system capable of handling both non-alpha textures of high quality and alpha-textures with four levels of alpha.

Hardware layout for PACKMAN-punch-through
Fig. 7 illustrates a corresponding hardware layout for PACCCKMAN combined with PACKMAN-punch-through.

Fig. 8 is an example of a hardware implementation of a combined bit extender that can generate the 8-bit color from either a 3-bit quantized color representation or from a 4-bit quantized color representation.

Just as well as combining PACCCKMAN with PACKMANalpha or PACKMAN-punch-through, we can combine PACKMAN (with 4-bit intensity codeword or table) with PACKMANalpha or PACKMAN-punch-through. In such schemes, it would be good to modify the bit layout of the 4-bit PACKMAN (4P) discussed above and illustrated in Fig. 9 so that the colors are in

21

corresponding place as for PAP, PAA, PAT, PPP and PPT. Fig 10 shows such a layout, here denoted Modified-4-bit-table-PACKMAN, or M4P for short. Thus, one way of combining PACKMAN with PACKMANalpha would be to use M4P for alpha free textures, and PAP, PAA and PAT for textures with alpha.

5   Similarly, we could combine PACKMAN with PACKMAN-punch-through by using M4P for alpha free textures, while using PPP and PPT for textures with alpha.

In this application, several of the modes have the same number of bits for

10   the intensity codeword. This means, in some cases, such as in the PP and PAP mode, that they can share the same codebook, which preserves memory. However, often quality can be enhanced by having different codebooks for two modes even if they are the same size. For instance, the PP mode will be used in conjunction with the PC mode. An optimal codebook will use this

15   fact not to spend codebooks entries for blocks that the PC handles well. The PAP mode, which has exactly the same bit layout as the PP mode, will not be used in conjunction with the PC mode, and the optimal table might therefore be different.

20   An even bigger difference is that between the PP/PAP modes that use 3x4 bits for RGB codeword and the PAA/PAT modes that use 3x3 bits for RGB. Here we also expect the optimal table to be different. Therefore we have control signals ("is PAP", "is PAA/PAT") going into the codebook lookup block so that the correct codebook can be used.

25

It should also be noted that it is possible to use the same codeword for all modes. This will give slightly worse quality but a smaller hardware solution.

In the case where we combine ordinary PACKMAN (4bit intensity codeword) instead of PACCCKMAN (3 bit intensity codeword), we may need a different table for the PACKMAN mode since it will be of a different size.

It will be understood by a person skilled in the art that various modifications and changes may be made to the present invention without departure from the scope thereof, which is defined by the appended claims.

REFERENCES

5   [1]   E. J. Delp and O. R. Mitchell, "Image compression using block truncation coding", *IEEE Transactions on Communications*, Vol. COM-2, No. 9, pp. 1335-1342, September, 1979

10   [2]   G. Campbell, T. A. DeFanti, J. Frederiksen, S. A. Joyce, L. A. Leske, J. A. Lindberg and D. J. Sandin, "Two bit/pixel full color encoding", *SIGGRAPH '86 Conference Proceedings*, Vol. 20, No. 4, pp. 215-223, August, 1986

15   [3]   US Patent No. 5,956,431

[4]   T. Akenine-Möller and J. Ström, "Graphics for the masses: A hardware architecture for mobile phones", *ACM Transactions on Graphics*, Vol. 22, No. 3, *Proceedings of ACM SIGGRAPH 2003*, pp. 801-808, July, 20   2003

[5]   S. Fenney, "Texture compression using low-frequency signal modulation", *Graphics Hardware 2003*, pp. 84-91, July 2003

25   [6]   Swedish Patent Application No. 0303497-2

CLAIMS

1. A method as disclosed by the present invention.

2. A system as disclosed by the present invention.

5

3. A signal representation of an image block as disclosed by the present invention.

10   ---

24

ABSTRACT

The present invention discloses an enhanced image compression or encoding method and system and the corresponding decompression or decoding method and system.

5

(Fig. 6)

1/10
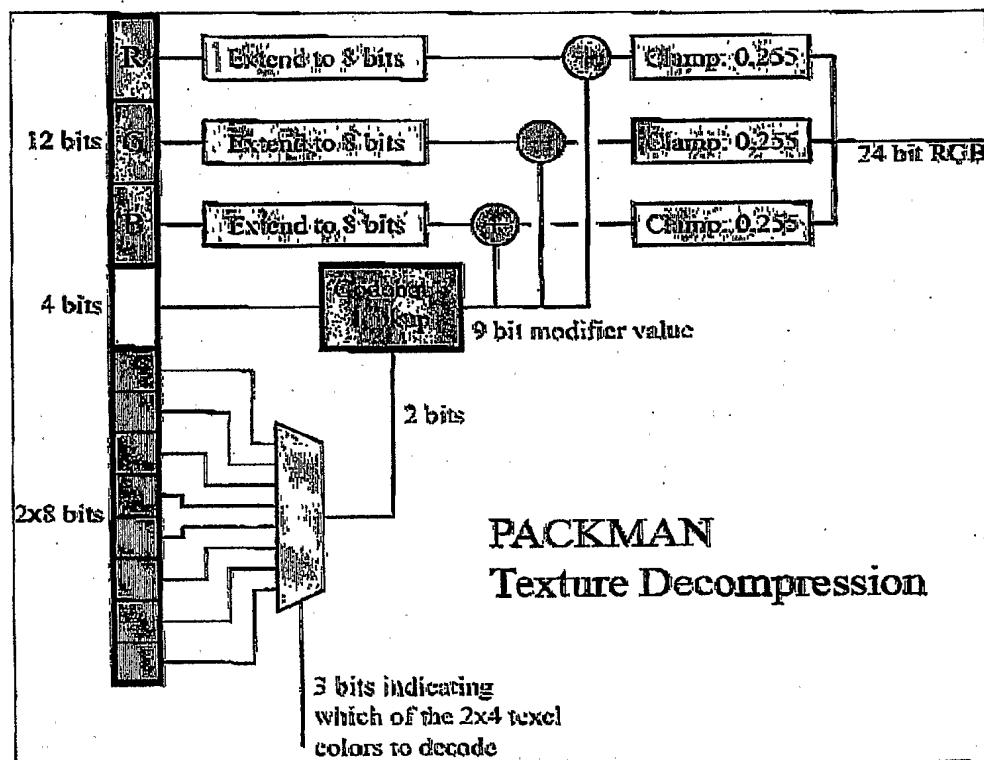
PACKMAN
Texture Decompression

Fig. 1

2/10

Bit layout for a PACCCKMAN texture (alpha free texture)

PACCCKMAN-PACKMAN block (PP)
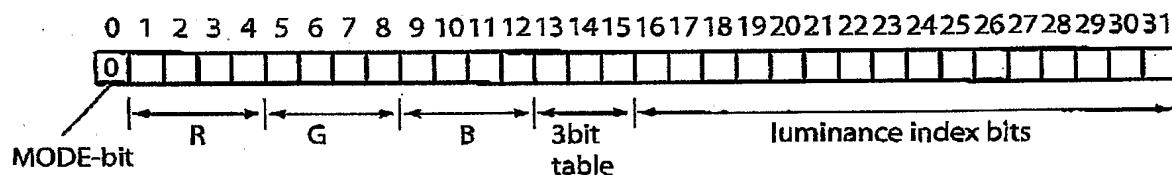
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

MODE-bit   R   G   B   3bit table   luminance index bits

PACCCKMAN-CCC block (PC)
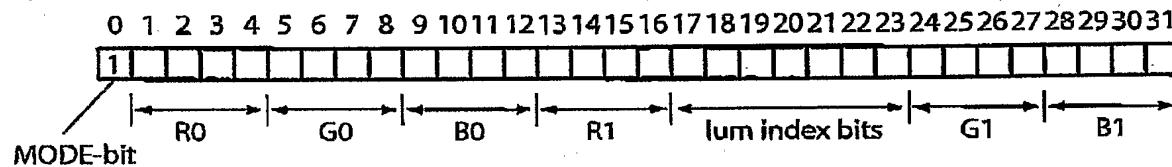
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

MODE-bit   R0   G0   B0   R1   lum index bits   G1   B1

Fig. 2

3/10

Bit layout for a PACKMANalpha texture (alpha texture)

PACKMANalpha-PACKMAN (PAP) block

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

MODE-bit

|← R →|← G →|← B →|← 3bit table →|← luminance index bits →|
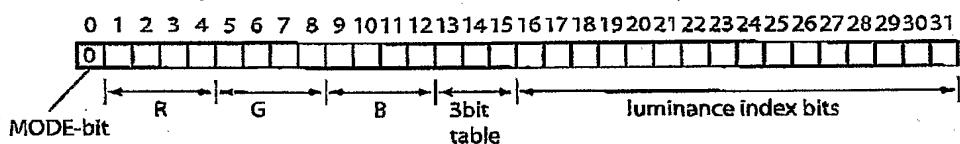
PACKMANalpha-ALPHA (PAA) block
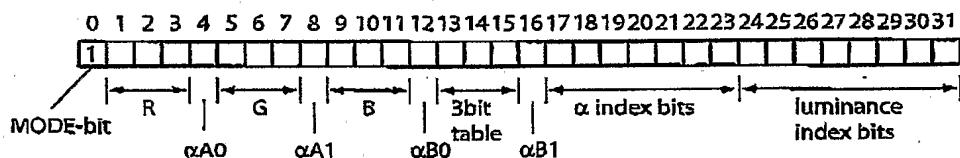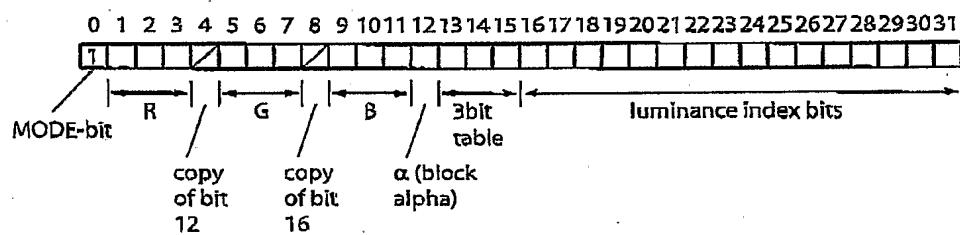
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

MODE-bit

|← R →|← G →|← B →|← 3bit table →|← α index bits →|← luminance index bits →|

αA0   αA1   αB0   αB1

PACKMANalpha-TWOTIMER (PAT) block

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

MODE-bit

|← R →|← G →|← B →|← 3bit table →|← luminance index bits →|

copy
of bit
12

copy
of bit
16

α (block
alpha)

Fig. 3

4/10

Bit layout for a PACKMAN-Punch-through texture (alpha texture)

PACKMAN-Punch-PACKMAN (PPP) block
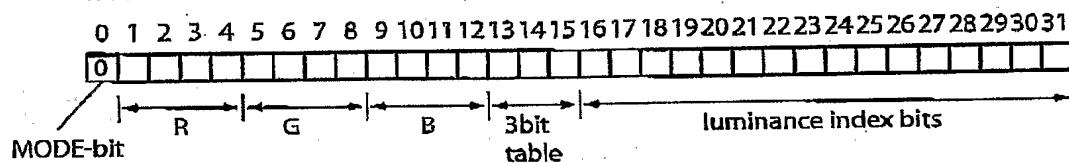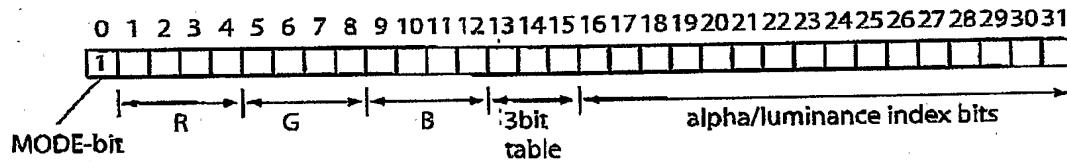
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

MODE-bit   R   G   B   3bit table   luminance index bits

PACKMAN-Punch-Through- (PPT) block

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

MODE-bit   R   G   B   3bit table   alpha/luminance index bits

Fig. 4

5/10

Fig. 5

6/10

Fig. 6

7/10

Fig. 7

8/10
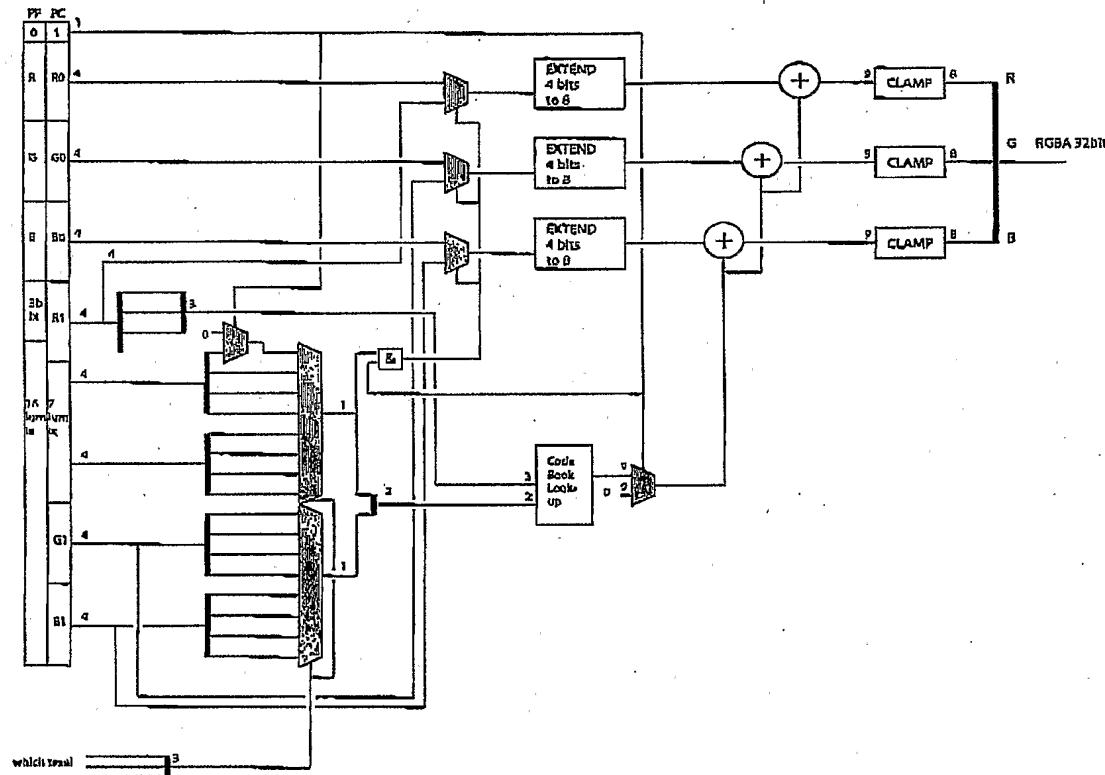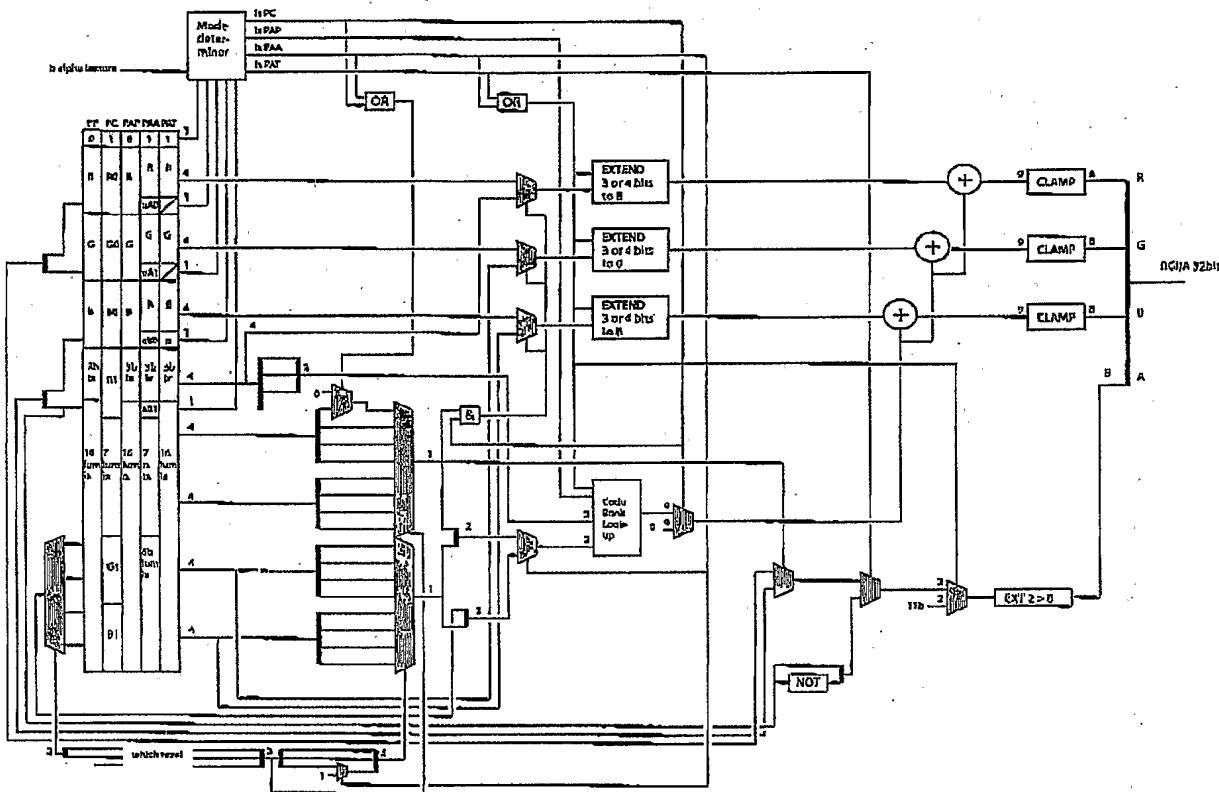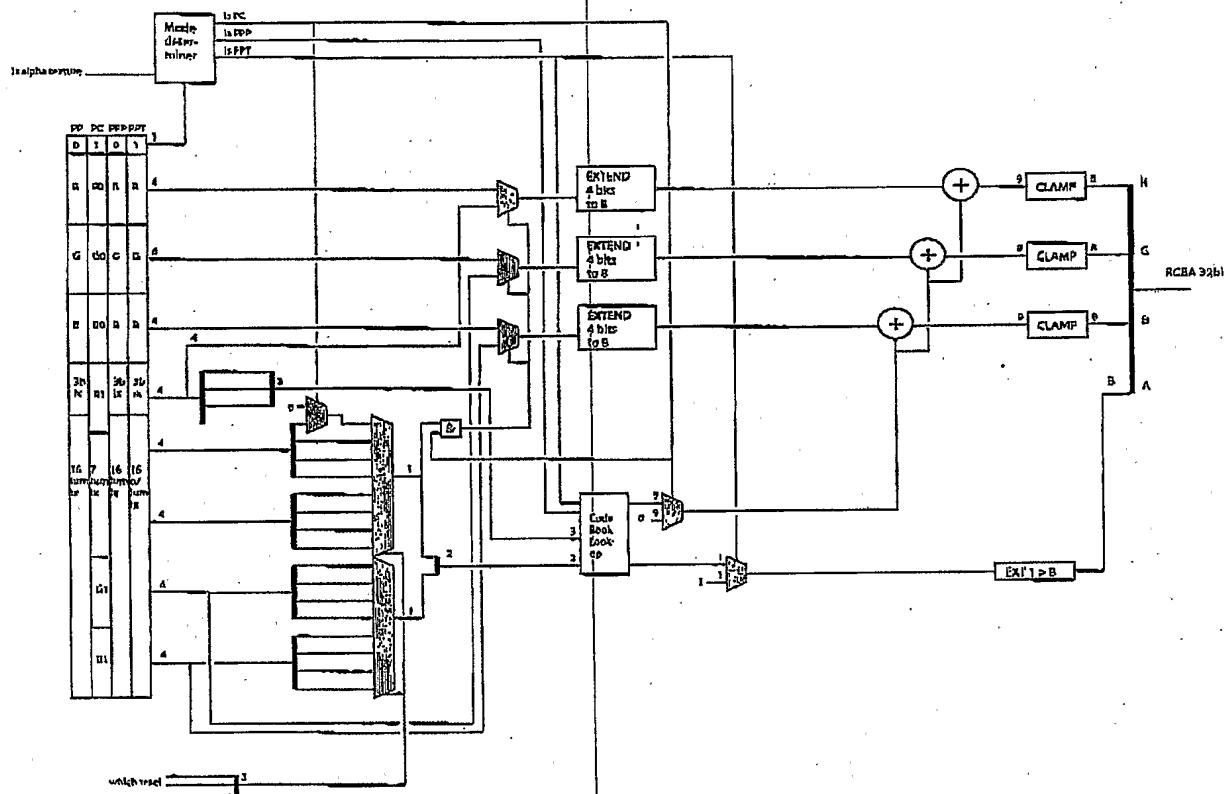
extend 3 or 4 bits to 8

Fig. 8

9/10

Bit layout for an 4-bit-table PACKMAN texture

4-bit-table-PACKMAN block (4P)
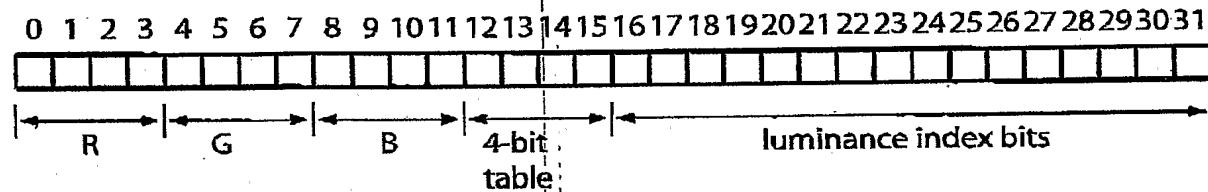
Fig. 9

10/10

Bit layout for an MODIFIED-4-bit-table PACKMAN (M4P) texture (alpha free)

MODIFIED-4-bit-table-PACKMAN block (M4P)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

|←——— R ———→|←——— G ———→|←——— B ———→|bit 0-2|←———————— luminance index bits ————————→|

bit 3
of 4-bit
table

bit 0-2
of 4-bit
table

luminance index bits

Fig. 10